Overview

1. Boolean Operations and Truth Tables

2. Nested Loops

3. Next Time
Last Time

Last time we discussed control flow and loops. As a refresher:

- The `for` and `while` loops are essential for doing tasks repeatedly, often while incrementing a value.
- There exists a `switch` statement which gets used in place of nested `if` statements, often when there are many cases to consider.
- Infinite loops happen, `break` statements come in handy, and we’ve vaguely been seeing functions in use but haven’t actually discussed them.

Today, we will see nested loops, as well as revisit Boolean operations more formally by presenting so-called Truth Tables. Buzzword: DeMorgan’s Laws.
Last time we discussed control flow and loops. As a refresher:

- The `for` and `while` loops are essential for doing tasks repeatedly, often while incrementing a value.
- There exists a `switch` statement which gets used in place of nested `if` statements, often when there are many cases to consider.
- Infinite loops happen, `break` statements come in handy, and we’ve vaguely been seeing functions in use but haven’t actually discussed them.

Today, we will see nested loops, as well as revisit Boolean operations more formally by presenting so-called *Truth Tables*. Buzzword: DeMorgan’s Laws.
1 Boolean Operations and Truth Tables

2 Nested Loops

3 Next Time
Boolean Operations

Boolean operations is just a fancy name for functions which accept 0’s and 1’s as inputs, and then output either 0 or 1. These are important for coding since often we will want to change the direction of our program based on a true/false condition, create switches for whether or not an output should be printed, and so on.

Complicated Boolean functions exist, but are not relevant to us here. For the most part, we are only going to worry about using and, represented by &&, and or, represented by ||.
Recall: Logical Operators

From last week, recall that C++ has logical operators implemented. These operators take in two boolean values (true/false values, 0/1 values), and return a single boolean value. For example:

- `||` means `or`, `&&` means `and`
- `true || false` will return `true`
- `true && false` will return `false`
- `false || false` will return `false`
- `false && false` will return `false`
- `true || true` will return `true`
- `true && true` will return `true`
Truth Tables

- Truth tables are a way to visualize the Boolean functions which take in two values (denoted $p$ and $q$ usually), and outputs a single Boolean value.

- Since there are only 4 possible combinations of inputs, these tables are 2 by 2.

- Note: In everyday language, “or” tends to mean exclusive or (XOR), since its usually “one or the other”.

- Don’t memorize these, you’ll absorb them by osmosis if you’re doing your work.
Here is a short example of how these operators can be used in control flow:

```cpp
using namespace std;

int main()
{
    bool feelingInspired = true;
    bool TooMuchToDo = true;  // Times 10
    bool NothingToDo = false;
    if( (feelingInspired) || (TooMuchToDo)){
        do_work();
    } else if ( (feelingInspired) && (NothingToDo) ){
        renew_lost_hobbies();
    } else {
        ask_why_notInspired();
    }
    return 0;
}
```
DeMorgan’s Laws

The one operation that hasn’t been discussed, but is relatively intuitive, is the not operation. This operation negates a Boolean value, or flips it from 0 to 1, 1 to 0, true to false, false to true. In c++, this operation is denoted by !, so asking the computer,

\[ \text{true == !false} \]

will return true. Closely associated with this operator are the so-called “DeMorgan’s Laws”, which can be written as

\[
\text{not (A or B) = (not A) and (not B)} \\
\text{not (A and B) = (not A) or (not B)}
\]

How useful are these laws? Depends on what you’re coding. Some people rarely have a need for them, others probably use them all the time.
Try this exercise on your own:

// Write these without negations!
!(x!=5 && x!=7)
!(x<5 || x>=7)
!( !(a>3 && b>4) && (c != 5))
Try this exercise on your own:

```cpp
// Write these without negations!
!(x!=5 && x!=7)
!(x<5 || x>=7)
!( !(a>3 && b>4) && (c != 5))
```

```plaintext
// solutions
x == 5 || x == 7
x >= 5 && x < 7
(a>3 && b>4) || (c == 5)
```

Useful? Its definitely a good practice to write such statements as simply as possible for others to understand. Plus, you’ll probably get tested on them, so, **useful**.
1. Boolean Operations and Truth Tables

2. Nested Loops

3. Next Time
Sometimes one isn’t enough. By nesting loops inside each other, we are able to do things like iterate over a 2D table of numbers, read through a folder of files and each line in each file, and so on. An easy example of nested for loops is:

```cpp
#include <iostream>
using namespace std;

int main() {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 3; j++) {
            cout << i * j << " ";
        }
        cout << endl;
    }
    return 0;
}
```
Nested Loops: Another Example

Here is an example of nested while loops (less common):

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string keyWord1 = "Thomas"; string keyWord2 = "Merkh";
    string sentence = "The TA Thomas comes from the Merkh family";
    bool word1found = false; bool word2found = false;
    while( !word1found ){
        while( !word2found ){
            // iterate over sentence string searching for word2
        }
        // iterate over sentence string searching for word1
    }
    return 0;
}
```

Side Note: As written, these loops would be dangerous. What would happen if the sentence didn't contain either my last or first name?
If we are using a for loop which iterates over \( N \) objects or takes \( N \) steps, then we say that the loop is \( \mathcal{O}(N) \), or “big O” \( N \). This notation does have an exact mathematical meaning, but we can just take it to mean that this part of the program does \( N \) computations.

If we nest two for loops which iterate \( N \) and \( M \) times respectively, what do you think the algorithmic efficiency would be?
Efficiency and Algorithms

If we are using a for loop which iterates over $N$ objects or takes $N$ steps, then we say that the loop is $O(N)$, or “big O” $N$. This notation does have an exact mathematical meaning, but we can just take it to mean that this part of the program does $N$ computations.

If we nest two for loops which iterate $N$ and $M$ times respectively, what do you think the algorithmic efficiency would be?

The answer is $O(NM)$. This notation is useful for comparing algorithms and how fast they accomplish a task. For more, see https://en.wikipedia.org/wiki/Algorithmic_efficiency.
Curse of Dimensionality

As introduced in the previous slide, algorithmic efficiency is something to consider when writing longer codes. For example, when I was an undergraduate, I developed physics simulations in C++ which had to:

- Loop over $T$ time step
- At each time step, loop over the $x$, $y$, and $z$ coordinates

Assuming that there were $N$ points in each dimension and $N$ time steps, this algorithm would've had efficiency $O(N^4)$. Therefore, even if there were only $N = 100$ points in each direction, the program would need to do $100,000,000$ (one hundred million) computations.

The buzzword for this is the curse of dimensionality, since each time another dimension is added, this program would take at least 100 times longer to complete.
As introduced in the previous slide, algorithmic efficiency is something to consider when writing longer codes. For example, when I was an undergraduate, I developed physics simulations in C++ which had to:

- Loop over $T$ time step
- At each time step, loop over the $x$, $y$, and $z$ coordinates

Assuming that there were $N$ points in each dimension and $N$ time steps, this algorithm would’ve had efficiency $O(N^4)$. Therefore, even if there were only $N = 100$ points in each direction, the program would need to do 100,000,000 (one hundred million) computations.

- The buzzword for this is *the curse of dimensionality*, since each time another dimension is added, this program would take at least 100 times longer to complete
Pseudocode is a way to communicate an algorithm or program, which is simplified and strips away any language-specific syntax. This is useful when trying to reason out a particularly tricky code, communicate to another what a given code is doing, or to access the algorithmic efficiency of a program. For example, consider the following pseudocode:

```
for each folder in current directory
    open the folder
    for each file in the folder
        delete file
    close folder
    delete folder
print "Cleared current directory"
```

Easier to understand than what the actual c++ would look like.
Challenge Question

Problem

Given a string of comma-separated numbers, find the substring with the maximum sum.

Example (Solved for you)

string closingPrices = "-4,-10,2,8,-1,5,7,-7,5,-10,-10,-10,0"
The substring with the largest sum would be "2,8,-1,5,7".

Example (Unsolved)

string closingPrices = "5,-3,3,8,-2,6,-10,13,1,-12,15,-9,-10"
Task: Come up with an algorithm for how you would solve this problem, and write this down using pseudocode (don't actually code it up, unless you're interested).
Challenge Question

Problem

Given a string of comma-separated numbers, find the substring with the maximum sum.

Example (Solved for you)

string closingPrices = 
"-4,-10,2,8,-1,5,7,-7,5,-10,-10,-10,0"

The substring with the largest sum would be "2,8,-1,5,7".

Example (Unsolved)

string closingPrices =
"5,-3,3,8,-2,6,-10,13,1,-12,15,-9,-10"

Task: Come up with an algorithm for how you would solve this problem, and write this down using pseudocode (don't actually code it up, unless you’re interested).
Problem

Given a string of comma-separated numbers, find the substring with the maximum sum.

Hint: Nested loops would be helpful here!
**Problem**

*Given a string of comma-separated numbers, find the substring with the maximum sum.*

Hint: Nested loops would be helpful here!

Here is one possible solution which checks every possible substring:

initialize largestSum to zero
initialize the string bestString
for each number in the original string, call it x:
  for each number that comes after x, call it y:
    compute sum of numbers between x and y
    if this sum is > largest sum:
      set largestSum equal to this sum
      set bestString equal to string from x to y
print out bestString
Questions to think about:

- Is that the most efficient method of finding the best substring?
- How many sums need to be tested for the method proposed? This is essentially its algorithmic efficiency.

These are bonus questions of course, but also relevant. I took this example straight from an internship interview.
1 Boolean Operations and Truth Tables

2 Nested Loops

3 Next Time
Next Time

- Do-While loops, Summary of flow control structures, Random numbers
- Office Hours this week: 11:30 - 1pm (If no one is present at 12:30 and no one warns me that they’ll be attending, I will leave at 12:31).